CS475 - Assignment 6

(KDTree Redux)

REWARD: 200 points

DUE: Fri Apr 8 at 11pm PT

TEST DATA: testDataA6.tar or testDataA6.zip
LIBRARY: mat.tar, mat.zip
NEW The 7.4 library includes an extra assert that might catch some mal-formed requests to sort by a column.

NEW This is the almost the same assignment as assignment 5. I made some changes based on our discussion in class. I want some extra output. Changes in the text are noted with the new arrow icon. This is graded as if a new assignment. It does not replace the assignment 5 score but gives you a chance to get more points and reduces the pain of the A5 if you didn't do as well as you would have liked. A5 will be worth 100 points and together they will be worth 300. **If you got the last assignment perfect** then just add the noted prints below and resubmit.

1 The Task

It is usually the case that you can consider the training data to be exemplars of correct answers and so points that are similar to a training data point might reasonably be assumed to have the same output value as that training data point. We'll call the following the Rule of Similarity:

if **f(x)** $\rightarrow$ **y** then **f(z)** $\rightarrow$ **y** for **z** near **x**.

As practitioners we have several choices to make. 1) what points to we pick? and 2) what do we mean by distance? In the first case, we have some tools we have discussed already for picking points. SVMs, for instance, try to pick points along the boundaries know to delineate different values of **y**. If the dimensions are somewhat large and we don't have a lot of training points or the boundary is very non-linear then perhaps we just use all the training points. If we have too many dimensions then we can try to reduce the dimensions with PCA first and then use K nearest neighbor (KNN).

For distance we want a meaningful distance where the the Rule of Similarity holds. For the KD-Tree algorithm we will want a true distance measure in which the Triangle

Inequality holds. The ”go to” distance measure is Euclidean distance which is in the family of Lebesgue norms or Lp norms or just p-norms. These norms range from taxi cab distance to the max of the magnitude of all distances. The similarity measure BC or Bhattacharyya coefficient can be used to create a distance measure: sqrt(1 - BC(x, y)) where x and y are vectors. Angular Distance function may have an application for some problem you are trying to solve in that is computes the angle between two points measured at the origin you chose. This may be particularly useful in higher dimensional space where Lp Norms tragically lose useful meaning.

Implement a C++ program called **kdtree** to find the single nearest neighbor a query item. It should use the matrix library trick shown in class where you create a tree of row vectors in a matrix by sorting subsets of rows in a specific columns. The result is a kdtree in a matrix. The algorithm is the same as kdtree in the book but the tree is built without any node object and all inside a matrix by just reordering the rows of the matrix.

Here is a example of a tree stored as an array:

```
0   |
  1 |  left sub tree
2   |
    3  <-  root location is = (total length)/2
4   |
  5 |  right sub tree
6   |
```

Each parent node is discovered as a middle (median) between the two halves of the data when the data is sorted on a feature column. The tree begins by being sorted in column 1 (for a labeled matrix). Then the middle point is chosen and the data above and below that middle point is then sorted on column 2. Those halves are then split and the data above and below that midpoint are sorted on column 3, then back to column 1 etc. This builds the equivalent of the KD-Tree. Now the kdtree algorithm is simply the C++ version of kdtree.py but using this tree-in-a-matrix data structure instead of lists and dictionaries. Turns out to be fairly efficient and easy to implement.

Things to watch out for:

- In labeled matrices like what we have for the color data uses the first column to label the rows. See the matrix library for details. Be sure you don’t use column 0 as the key in distance or sorting when creating the KDTree. But be sure you use the default that maintains the label with each row.
- Be sure to use the best distance found in one half of the tree in the call to search

the other half. That is the best distance information doesn't just flow up the tree it follows the traversal. This will make your algorithm faster.

- Use only the latest matrix library (version 7.3 or later).

Input is first a labeled matrix of rows where each row begins with a word (no whitespace in the word!) followed by d columns of data. The matrix can be read by the readLabeledRow() function which returns a list of symbols in an specialized object. The first column in the read matrix is the index of the corresponding label. That is, it "hashes" the string to numbers and puts the numbers in a numeric array. Not ideal, but workable.

NEW To help with debugging: print out the KDTree matrix using the printLabeledRow method.

The second half of the input is a matrix of unlabeled data of d columns each. For each row in the matrix use the kdtree algorithm to look up the closest answer and print out that label. Sample output will be provided. In some cases there is more than one closest answer (see testData for notes equivalent answers in a txt file). Any of the equivalent answers is acceptable.

2 Implementation Help

We have a lot left to do this semester so here is the outline I explained in class:

```
The KD-Tree algorithm in C++ comments

// BUILD THE KD-TREE
// sort by column c
// median is middle of sorted list
// call build with left side and c+1
// call build with right side and c+1

// SEARCH THE KD-TREE
// kdtree(int bestrow, int bestdistance) returns bestrow and bestdistance
// if leaf node
//    INCREMENT COUNTER
//    if better bestrow save that as new best node

// else if parent node case
// if item left of split point?
```

```
//     do left then right
//        call search(best) on left -> bestrow and best
//        if dist(item, split point) in dimension c > best then return
//        call search(best) on right
// else
//     do right then left
//        call search(best) on right -> bestrow and best
//        if dist(item, split point) in dimension c > best then return
//        call search(best) on left

// do split point (parent)
//     INCREMENT COUNTER
//     if better bestrow save that as new bestnode
```

Your algorithm must be the KD-Tree algorithm and not rely on a brute force search. Hint: you only need to check for a new better distance at the leaf and at the split (parent). Sometimes you don't even check the parent because of the short cut test (their column c value is too far away). Keep a counter of how many color comparisons are made. See the two places where the increment counter is in the algorithm above. Print the counter after each search in the format shown. For example:

```
FIND: 197 227 132
Num Compares: 7
8.60233 MediumSpringBud 201 220 135
```

3 Submission

Homework will be submitted as an **uncompressed** tar file to the homework submission page linked from the class web page. Be sure to include a make file to build your code and do NOT include the picture files. I will supply some. You can submit as many times as you like. **The LAST file you submit BEFORE the deadline will be the one graded.** For all submissions you will receive email at your uidaho.edu mail address giving you some automated feedback on the unpacking and compiling and running of code and possibly some other things that can be autotested.

Have fun.